CS 161 Project 2 Design Doc (Phish Phighters)

Data structures - We are going to organize data on Datastore in the form of structs corresponding to users, files, intermediates and sharehubs. We'll ensure confidentiality and integrity by encrypting the data and then MACing it before storing it. Also, each file will be broken down into a linked list where each node points to a corresponding ciphertext and also the next node.

Helper functions:

- EncryptThenSign: Encrypts and then signs the data.
- VerifyThenDecrypt: Verifies the signature and then decrypts the data.
- GenerateUserUUID: Generates a UUID for a user.
- GenerateUserKeys: Generates encryption and verification keys for a user.
- GenerateBodyKeys: Generates keys for encrypting file body and nodes.
- MacThenDecrypt: Verifies HMAC and then decrypts.
- EncryptThenMac: Encrypts and then applies HMAC.

User Authentication - if user is new, create UUID by hashing the username and taking the last 16 bytes of the hash. The key to encrypt the user struct is generated with Argon2Key(password, username$\oplus$0x5c) and the key to MAC the struct is generated with Argon2Key(password, username$\oplus$0x36). It is important that the UUID is deterministic; this means that GetUser will error iff the inputted username does not exist or if the keys generated by username and password to decrypt the user struct does not work..

Multiple Devices - when a user logs in, they will pull their user struct from Datastore using their UUID. Whenever they make a change, they edit their local copy and upload it to Datastore. Whenever they need to access any information from the struct, they will pull the struct from. Because the user is always pulling from Datastore and always updating to Datastore, it is impossible for their devices to be out of sync.

File Storage and Retrieval - Every file will be uploaded to Datastore in the form of a linked list. The plaintext is split into **n** different "blocks", depending on the length of the plaintext. The key for block **i** is HashKDF(**α**,**i**) for a randomly generated integer **α**. Each block is also encrypted with a unique, random IV and MACed with the key Hash(**α**$\oplus$**i**), and then uploaded to a randomly generated UUID. Now, a node struct is uploaded to Datastore to a random UUID. This struct contains the UUID (as a pointer) of the ciphertext we just uploaded and the UUID (as a pointer) to the next node struct of the list (unless it is the last node). The node struct at the head of the list will also contain a pointer to the end of the list and **n**, the length of the list as well as the first node in the list and the 2 random numbers for the file **α** and **β** (this first node is also called the Filehead). Each node struct is encrypted-then-MACed with HashKDF(**β**,**i**) and random IV and Hash(**β**$\oplus$**i**) for a randomly generated **β**. These **α** and **β** are stored in the user struct for the information that the users have on the files that they own, and are indexed under the UUID of the head of the linked list.

To retrieve the file, the user starts at the Intermediate which is decrypted with RSA which contains a pointer to the filehead and the keys for decryptign the headfile. The headfile is decrypted with symmetric keys found in the intermediate. Then we compute the keys using **β** to MAC-then-decrypt the node. Then, it can use the node and **α** to MAC-then-decrypt the ciphertext and then traverse to the next node and repeat, calculating the keys along the way by keeping track of its index along the linked list.

Efficient Append - Because files will be stored as a linked list data structure, we can easily append to the end by accessing the final element. The first element is constant lookup time because it is stored in the user struct and the final element is constant lookup time because its UUID is stored in the headfile which is the start of the list. We would download only these 2 blocks to change the UUIDs stored in them to reflect the new block that we are adding to be the new final block.

File Sharing - The CreateInvitation function in the system is designed for initiating the process of file sharing. It begins by verifying whether the user attempting to share a file is indeed the owner of that file. If the user is the owner, the function creates a ShareHub structure. This structure contains crucial elements like the UUID of the file head, randomly generated numbers α and β (used for decryption and MAC verification of the ciphertexts and the linked list), and the UUID of the recipient. Once this structure is assembled, it is encrypted using the public key of the intended recipient and signed with the private key of the sender, ensuring both confidentiality and authenticity. Additionally, the function updates the owner's user structure by adding the UUID of this ShareHub to the SharedByMe list, effectively keeping a record of all files shared by the user. In cases where the user is not the owner of the file, the function instead shares an existing invitation, specifically the Intermediate structure, thereby avoiding the creation of a new invitation.
The AcceptInvitation function complements this process by enabling users to accept shared files. When a user receives an invitation, this function retrieves the shared structure, which could be either a ShareHub or an Intermediate, using the provided UUID from the Datastore. The authenticity of this invitation is verified through a digital signature check using the sender’s public key. Post verification, the recipient decrypts the share structure using their private key. The function then proceeds to create an Intermediate structure embedded with information from the decrypted share structure, which includes essential pointers and keys for file access. This new structure also flags the recipient as a non-owner of the file (IsOwner set to false). The UUID of this newly formed Intermediate structure is then added to the SharedToMe list in the accepting user’s struct. Lastly, the Intermediate structure is serialized, encrypted, and signed before being stored in the Datastore under a UUID that is derived from the accepting user's username and the filename, thereby completing the file sharing process.

File Revocation - The `RevokeAccess` function in the system is designed to revoke a user's access to a shared file. This process begins by the function re-encrypting the file in question, which effectively alters the file's access credentials. This re-encryption step is crucial as it

ensures that the revoked user no longer has the necessary keys to access the file. Once re-encrypted, the function then focuses on the management of shared access information.

The function identifies and accesses the `ShareHub` corresponding to the revoked user and replaces its content with non-relevant or 'garbage' data. This action renders the previously shared access information useless. Simultaneously, the revoked user is removed from the `SharedByMe` list of the owner. This is an important step in ensuring that the system's records reflect the updated access rights.

Furthermore, the function proceeds to update the remaining share structures associated with the file. This is done by iterating through the `ShareList` and modifying the `ShareHub` for each user who still has access to the file. Each of these `ShareHub` structures is updated with new parameters, including the new UUID of the re-encrypted file and the updated α and β values, which are necessary for decryption and MAC verification. This ensures that all legitimate users maintain access to the file with the updated security parameters, while the revoked user is effectively barred from accessing the new version of the file.

In essence, the `RevokeAccess` function systematically alters the file's encryption, invalidates the access information of the revoked user, and updates the access information for all other users, thereby maintaining the integrity and confidentiality of the shared file in the face of access revocation.

UUID	Encrypted	Key Derivation	Value at UUID	Description /Relationship
UUID.frombytes(sha256(username)[:16])	Yes	Symmetric Key generated when user first signs up through their login credentials	RSA decryption key, sign key, files shared by me {user UUID : share UUID}, files shared to me {user UUID : share UUID}, files owned by me {head UUID : α and β}	This is a very important part of the program because this is where the user will constantly be pulling from to share files, revoke access, store files, etc.
UUID.New()	Yes	Symmetric key randomly generated	UUID of file, shared by, shared to, α and β	This share struct is to be shared and stored in the user structs of people.
UUID.New()	Yes	Symmetric key randomly generated, derivation in *File Storage and Retrieval*	UUID of ciphertext location, UUID of next node (null if last node), UUID of last node (null if not head), n (length of file, null if not head)	File linked list struct - files will be split as a linked list of structs. Each struct will have the encrypted contents of the “block”, the UUID of the next block, and in the case of the head of the list, the UUID of the last block.
UUID.New()	Yes	Symmetric key randomly generated, derivation in *File Storage and Retrieval*	Encrypted ciphertext	This is pointed to by the linked list struct